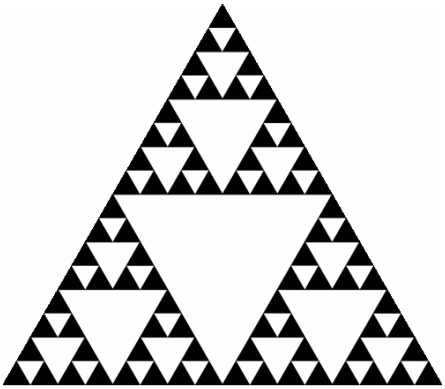
SE assignment **Univ. Rennes 1**

Sierpinski triangle

The Sierpinski triangle is a *fractal* and attractive fixed set with the overall shape of a triangle, subdivided recursively into smaller triangles.



Notice that this picture only shows us an *approximation* of the Sierpinski triangle.

We have the following general procedure for the generation of the Sierpinski triangle, from any given image. Take the pixel matrix representing the original image, shrink it to half height and half width, and make three copies of it. Then, position these three copies in an image having the same size of the original, by following the three rules:

- one copy is placed so that its lower left-most corner coincides with the one of the image;
- one copy is placed so that its lower right-most corner coincides with the one of the image;
- one copy is placed in the upper remaining space, at its center, by avoiding any overlaps with the two previous copies.

This procedure can be iterated, in theory, to infinity. In practice, it is possible to stop it as soon as no changes on the current image are anymore visible on our computer screen. The Java class `TextGraphics` contains the `Sierpinski1` method that performs, in sequential, one iteration of this construction procedure.

Our settings

We will perform this assignment with a *virtual machine* capable to run as many threads as we like. However, since the computers we'll be using for this assignment are rather "real" and actually give a limit on the number of threads that we can run (the limit is actually given by the number of cores, because running two threads on the same core is not really useful ...), we will simply write our program in sequential, but we will design it in such a way that we could easily adapt it for our virtual multi-threading settings.

In the Sierpinski algorithm, we will suppose that every pixel of the image is assigned to a different thread. We will also suppose that two identifiers, named `i` and `j`, are assigned to every thread. To make things easier, we will work with black-and-white images, where the two possible "colors" (the white and the black) are represented by a Boolean value (`true` or `false`). We will suppose that the pixel matrix has size `n`×`m`, where both `n` and `m` are multiple of 4.

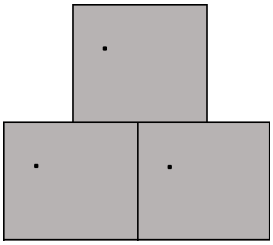
Notice that, from a practical point of view, when the number of threads is large but not enormous, it is actually possible to realize these settings with *GPU devices*.

The "Sierpinski1" method

Before starting to code, please take the time to check the `TextGraphics` class, to understand its attributes, and in particular the `Sierpinski1` method.

Where am I

In order to write the multi-threading method in our special settings, the first question that requires an answer is the following: how can every thread find out where its own pixel is located? In fact, the pixel color in the new image strictly depends on this information.



In the drawing above, the gray rectangles represent the places where the copies of the original image are placed in the new constructed image, obtained with one iteration of the Sierpinski construction rule. Is it possible to find conditions on the indices `i` and `j` that allow the threads to verify in which of those rectangles they are located? Should we have a dedicated piece of code for every rectangle?

Which are my reference pixels

The color of every new pixel in the new image depends on the color of 4 pixels of the original image. Since we can only represent black (`true`) and white (`false`), we say that the new pixel will be black if *at least one* of its reference pixels is black; it will be white otherwise. In the drawing above, the three pixels marked in black share the same 4 reference pixels in the original matrix. How to identify these 4 reference pixels on the basis of the location (rectangle) of the target pixel in the new image?

The multi-threading program

A few lines have already been included in the `Sierpinski2` method, in the `TextGraphics` class. There are basically two `for` loops, which simulate in a way our multi-threading settings: simply keep in mind that these two for loops can potentially be replaced by a call to `n`×`m` parallel threads, but we will not do this for real. Simply write the code *inside* these two for loops, and suppose we are working with the thread having identifiers (`i`,`j`).

Important!

In order to match with our settings, we'll have very strong constraints on the method structure: we can only declare and define Boolean variables, and the final computed value (the pixel color) will have to be the result of *one unique Boolean expression*. No other additional instructions are allowed.